Under consideration for publication in Formal Aspects of Computing

# Generating Tests from B Specifications and Dynamic Selection Criteria[1]

J. Julliand, P.-A. Masson, R. Tissot and P.-C. Bué

LIFC, Université de Franche-Comté
16, route de Gray F-25030 Besançon Cedex France
{julliand, masson, tissot, bue}@lifc.univ-fcomte.fr

**Abstract.** This paper is about generating tests from dynamic selection criteria called test purposes, in addition to structural tests, obtained from static selection criteria. We present a method that re-uses a behavioral model and an abstract test concretization layer developed for structural testing, and relies on additional test purposes. We propose, in the B framework, a process of test generation that uses the symbolic animation mechanisms of LTG (Leirios Test Generator) based on constraint solving, and guided by the test purposes. We build for that a B model that is the synchronized product of a behavioral B abstract model and a test purpose described as a labelled transition system. We prove the correctness of this method, and show some experimental results obtained on the IAS case study. IAS is an industrial smart-card platform dedicated to the operations of Identification, Authentication and electronic Signature. Our experiments show that the tests obtained from test purposes are complementary to the structural tests.
**Keywords:** Model-Based Testing, Test Purpose, IAS Case Study.

## 1. Introduction

B models are well suited for producing functional tests of an implementation by means of a *model-based testing* approach [BJK+05, UL06]. This approach, as is described in Sec. 5 and illustrated by Fig. 8, proceeds by writing a *formal behavioral model* (M) of the expected functionalities of a system. This model is an abstraction of any real implementation, and is supposed to provide a reliable view of the implementation under test (IUT). By applying selection criteria, a test generation tool can automatically extract tests from the model. These tests are particular "executions" of the model. They are sequences of operation calls, with values of their parameters and their results as predicted by the model. The tests are abstract since they have the same level of abstraction as the model. They are concretized by a *concretization layer* (CL) to become executable on the IUT. Comparing the results returned by the IUT with the ones predicted by the model allows delivering a verdict of the tests.

Structural testing uses static (syntactic) selection criteria, essentially providing control flow and data

[1] Research partially funded by the French National Research Agency ANR (POSE ANR-05-RNTL-01001) and the Région Franche-Comté.
*Correspondence and offprint requests to*: J. Julliand, P.-A. Masson, R. Tissot and P.-C. Bué

coverage of the model. The tests exercise the functionalities of the system by directly activating and covering the corresponding operations. Industrial studies have proven the efficiency of the method to detect faults in an implementation (see for example [EFHP02, BLLP04]). Writing M and CL is an important effort, but the cost is justified by the possibility to automatically compute a great number of smart test cases, with M as an oracle. Nevertheless, static selection criteria appear to be insufficient to exercise the IUT in tortuous situations. We think for example of some scenarios of attack of systems requiring strong security guarantees. Our objective is to benefit from M and CL to compute some additional tests that use a particular scenario as a selection criterion.

The scenario can be described by means of a *test purpose* (TP), which we consider as a dynamic (semantic) selection criteria that orchestrates the successive calls of the operations of the model. The tests extracted from the model by means of a test purpose are sequences of operation calls corresponding to the scenario.

The context of this work is the test generation from B models[2]. We use LTG (Leirios Test Generator) [JL07], the test generator from Smartesting[3], to automatically extract abstract tests from a behavioral model written in B. LTG uses a constraint solver for computing the tests. LTG produces structural tests by applying static criteria to cover all the paths of the control structure of every operation. Moreover, it is possible to assist the generation of tests by providing LTG with sequences of operation calls that describe the shape of the expected tests. We have validated our approach on IAS, an industrial standard for smart cards.

Our main contribution in this paper is to define in the B framework a process that uses LTG for generating abstract tests, but with a dynamic selection criterion, provided to LTG in the shape of a set of sequences of operations, described by a TP. Also, we have performed experiments that show that these tests are new tests w.r.t. the ones obtained from static criteria.

We give in Sec. 2 some preliminary definitions to our work. IAS, the case study on which we have experimented our approach, is described in Sec. 3. Section 4 defines test purposes, and proposes a language dedicated to their expression. The model-based testing process with static criteria using LTG, as well as our process based on dynamic criteria, are introduced in Sec. 5. Section 6 describes how to combine a behavioral model and a test purpose to obtain a B model for the test generation. Our experimental results are given in Sec. 7. We conclude, compare our proposition to related works and expose some future works in Sec. 8.

## 2. Preliminaries

This section gives the background required for reading the paper, with respect to B in particular. We give general notions about B abstract machines. We define the notions of B trace and B execution. We also define the restrictions due to the targeted application class and to the context of test generation.

First introduced by J.-R. Abrial [Abr96], a B abstract machine defines an open specification of a system by a set of operations. Intuitively, an operation has a precondition and modifies the internal state variables by a generalized substitution. An operation is provided with a list of parameters and can return results.

We address a particular class of specifications. Our specifications are defensive, i.e. we assume that an operation terminates whenever it is invoked with well typed parameters. That means that we consider environments that respect a contract: they always call the operations with well typed parameter values. We also assume that any operation returns a *status word* (the term is borrowed from the smart card world) that codifies a report of its execution. Therefore in the remainder of the paper, operations are defined as in Def. 1.

**Definition 1 (Operation).** Let $S_i$ be a substitution. Let $sw_i$ be a status word and $p_i$ be a list of parameter names. Let $T_i(p_i)$ be a typing predicate on $p_i$. An operation named $op_i$ is defined as $sw_i \leftarrow op_i(p_i) =$ PRE $T_i(p_i)$ THEN $S_i$ END.

For defining a B abstract machine, we need to remind the reader of the notions of B predicates and B generalized substitutions. B predicates on a set of variables $x$ are denoted by $P(x)$, $R(x)$, $I(x)$, $T(x)$, ... In the remainder of this paper, the predicate $I(x)$ denotes an invariant and $T(p)$ denotes a typing predicate on the parameter variables $p$. When there is no ambiguity on $x$, we simply denote the predicates by $P$, $R$, $I$, ...

[2] This paper is a revised and extended version of a paper [JMT08b] previously presented at the ABZ'08 conference.
[3] `http://www.smartesting.com`, formerly Leirios Technologies.

We denote by $S$ the B generalized substitutions and by $E$, $F$, ... the B expressions. Expressions are typed as natural, boolean, set, function or relation. Relations between two sets $A$ and $C$ are denoted as $A \leftrightarrow C$. Total and partial functions are respectively denoted as $A \rightarrow C$ and $A \nrightarrow C$. A pair of elements related by a relation or a function is denoted as $a \mapsto c$. Given a substitution $S$ and a post-condition $R$ we are able to compute the weakest precondition $P$, such that if $P$ is satisfied, then $R$ is satisfied after the execution of $S$. The weakest precondition, defined in [Abr96], is denoted by $[S]R$. We denote by $\langle S \rangle R$ the expression $\neg[S]\neg R$, intuitively meaning that if $\langle S \rangle R$ is satisfied, then a computation of $S$ exists terminating in a state satisfying $R$. Given a B substitution $S$, a particular predicate denoted by $prd_x(S)$ defines the relation between the values of the state variables $x$ before the execution of $S$ and the values of the state variable $x'$ after the execution of $S$. $prd_x(S)$ is the before-after predicate of $S$. It is defined in Def. 2. B abstract machines are defined as in Def. 3.

**Definition 2 (Before-after predicate).** Let $S$ be a substitution. The before-after predicate $prd_x(S)$ is defined as $prd_x(S) = \langle S \rangle (x = x')$.

**Definition 3 (B Abstract Machine).** A B abstract machine $\mathsf{M}$ is a tuple $\langle x, I, \text{Init}, OP \rangle$ where

- $x$ is a set of state variables,
- $I$ is an invariant predicate over $x$,
- Init is a substitution called *initialization*,
- $OP$ is a set of operation definitions as in Def. 1.

We denote as $X_{\mathsf{M}}$ (where $X \in \{x, I, Init, OP\}$) a component of the B model $\mathsf{M}$. If there is no ambiguity on the model that is considered, we simply denote it by $X$. A model $\mathsf{M}$ defines a set $\mathcal{A}_{\mathsf{M}}$ of operation names and a set $\mathcal{P}red_{\mathsf{M}}$ of B predicates over the state variables $x$ of $\mathsf{M}$.

The test cases are finite executions. We first define the notion of *B trace* of a B abstract machine in Def. 4. Intuitively, a B trace is a finite sequence of operation names starting after the initialization.

**Definition 4 (B Trace).** Let $\mathsf{M} = \langle x, I, \text{Init}, OP \rangle$ be a B abstract machine. A trace is a finite sequence $\tau_{\mathsf{M}} = \text{Init}; op_1; op_2; \ldots; op_n$ where $op_i$ is the name of an operation ($\in \mathcal{A}_{\mathsf{M}}$) defined in $OP$ as in Def. 1.

Several executions can be associated to a B trace because, for any operation $op_i$, there are possibly several parameter values $v_i$ of $p_i$ that satisfy the typing predicate $T_i(p_i)$. As can be seen in Def. 5, an execution is an instance of a B trace with parameter values for every operation call that satisfy the precondition $T_i(p_i)$.

**Definition 5 (B Execution).** Let $\mathsf{M} = \langle x, I, \text{Init}, OP \rangle$ be a B abstract machine. Let $\tau_{\mathsf{M}} = \text{Init}; op_1; op_2; \ldots; op_n$ be a trace of $\mathsf{M}$. $\sigma_{\mathsf{M}} = (op_1(v_1), w_1); (op_2(v_2), w_2); \ldots; (op_n(v_n), w_n)$ is an execution associated to $\tau_{\mathsf{M}}$, denoted by $\sigma_{\mathsf{M}} \in Exec_B(\mathsf{M}, \tau_{\mathsf{M}})$, if there is a sequence of state variable values $u_0; u_1; u_2; \ldots; u_n$, a sequence of status words $w_1; w_2; \ldots; w_n$ and a sequence of parameter values $v_1; v_2; \ldots; v_n$ such that

- $[x' := u_0] prd_x(Init)$,
- for any $i \in 1..n$: $[p_i := v_i] T_i(p_i) \wedge [x, x', sw_i, p_i := u_{i-1}, u_i, w_i, v_i] prd_x(S_i)$.

Since we assume our specifications to be defensive (i.e. the preconditions are limited to typing predicates), there is at least one execution associated to a B trace if $T_i(p_i)$ is a satisfiable typing predicate. Thanks to that, we assume that the executions respect the preconditions, i.e. the environment (simulated by the test generator) always calls the operations with well-typed parameter values. In other words, the test generator chooses parameter values that satisfy the precondition, i. e. the typing predicate $T_i(p_i)$. Moreover, the operation call $op_i(v_i)$ from the state $u_{i-1}$ gives the new state variable values $u_i$ and returns the status word $w_i$. $u_{i-1}$, $u_i$, $w_i$ and $v_i$ satisfy the before-after predicate of $S_i$.

## 3. IAS Case Study

This work was done in the framework of the RNTL POSE project, that brings together industrial (GEMALTO, SMARTESTING, SILICOMP/AQL) and academic (LIFC/INRIA CASSIS project, LIG) partners. The aim of the project was the validation of the conformity of a system to its security policy, especially for smart cards.

Experiments have been made with a real size industrial application, the IAS platform. Prior to the project, a behavioral model in B had been written by the LIFC and SMARTESTING, from which structural

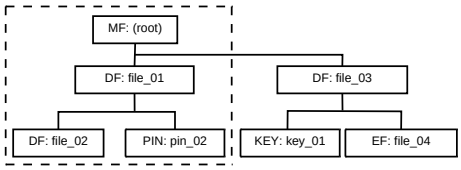


**Fig. 1.** A sample IAS tree structure

tests had been computed and executed on an IAS implementation by GEMALTO. We have extended these tests with tests computed from dynamic selection criteria.

IAS is a standard for Smart Cards developed as a common platform for e-Administration in France, and specified in [GIX04] by GIXEL. IAS provides services to the other applications running on the card. IAS conforms to the ISO 7816 standard.

The file system of IAS is illustrated with an example in Fig. 1. Files in IAS are either *Elementary Files* (EF), or *Directory Files* (DF), e.g. `file_01` and `file_02` in Fig. 1. The file system is organized as a tree structure whose root is designed as MF (*Master File*).

The *Security Data Objects* (SDO) are objects of an application that contain highly sensitive data such as PIN codes (e.g. `pin_02` in Fig. 1) or cryptographic keys that protect another data. They can be used to restrict the access to some of the application data.

The access to an object by an operation in IAS is protected by security rules based on security attributes. The access rules can possibly be expressed as a conjunction of elementary access conditions, such as *Never* (which is the rule by default, stating that the command can never access the object), *Always* (the command can always access the object), or *User* (user authentication: the user must be authenticated by means of a PIN code).

Let us present the variables of the model that we use in a forthcoming example of a test purpose given in Sec. 4.3. Let `X_ID` be a set of $X$ identifiers, where $X$ is either DF, PIN, OBJ or SDO. The variable `current_DF` ($\in$ `DF_ID`) stores the current selected DF. The variable `pin_02_dfParent` ($\in$ `PIN_ID` $\nrightarrow$ `DF_ID`) is a partial function that associates to a PIN the DF where it is located. The variable `rule_2_obj` ($\in$ `SDO_ID` $\cup$ {`always`, `never`} $\leftrightarrow$ `OBJ_ID`) is a relation that associates to a SDO the object that it protects. If the object is always (resp. never) accessible, then the SDO is replaced by the value `always` (resp. `never`). The variable `pin_authenticated_2_df` ($\in$ `PIN_ID` $\leftrightarrow$ `DF_ID`) is a relation that associates a PIN with the DF where the owner of the PIN is authenticated.

Consider for example the data structure shown in Fig. 1. The predicate `pin_02` $\mapsto$ `file_01` $\in$ `pin_02_dfParent` is true since the PIN object `pin_02` is located in the DF `file_01`. The predicate `pin_02` $\mapsto$ `file_02` $\in$ `rule_2_obj` is true if the access to the DF `file_02` is protected by a user authentication over the SDO `pin_02`. If `pin_02` $\mapsto$ `file_02` $\in$ `pin_authenticated_2_df` is true, then the access to the DF `file_02` is authorized, otherwise it is forbidden.

The services provided by the IAS platform can be invoked by means of various APDU[4] commands. Some of these commands allow the creation of objects: for example, `PUT_DATA_OBJ_PIN_CREATE` creates a PIN code, `CREATE_FILE_DF` creates a DF, ... Some are used to navigate through the file system, such as `SELECT_FILE_DF_PARENT` or `SELECT_FILE_DF_CHILD`. Some set the values of attributes: for example, `RESET_RETRY_COUNTER` is for resetting the PIN try counter to its initial value, `CHANGE_REFERENCE_DATA` is for changing a PIN code value, `VERIFY` sets a validation flag to `true` or `false` depending on the success of an authentication over a PIN code, ... Other commands are for changing the life cycle state of files, such as `DEACTIVATE_FILE`, `ACTIVATE_FILE`, `TERMINATE_FILE`, or `DELETE_FILE`, ...

In accordance with APDU commands, the IAS platform responds to a command by means of a status word (i.e. a codified number), which indicates whether the APDU command has executed correctly or not. If not, the status word indicates the nature of the problem that prevented the command from ending normally.

[4] *Application Protocol Data Unit* - it is the communication unit between a reader and a card; its structure conforms to the ISO 7816 standards

```
OP       ::=   operation_name
             |  "$op"
             |  "$op\{"  OPLIST  "}"

OPLIST   ::=   operation_name
             |  operation_name  ","  OPLIST

SP       ::=   state_predicate
```

**Fig. 2.** Syntactic Rules for the Model Layer

```
CHOICE   ::=   "|"  |  "⊗"
```

**Fig. 3.** Syntactic Rule for the Test Generation Directive Layer

## 4. Test Purpose

We see a test purpose as a means to exercise the system in a particular situation, for example w.r.t. a property. Based on his know-how, an experienced security engineer will imagine possible scenarios in which he thinks the property might be violated by an erroneous implementation. He describes the scenario as a test purpose.

We have defined in [JMT08a] a language to express such test purposes. It is based on regular expressions and allows the engineer to conceive its scenarios in terms of states to be reached and operations to be called. We present the language in Sec. 4.1. The starting non-terminal of its grammar is SEQ. We give its semantics in Sec. 4.2, and show a test purpose example in Sec. 4.3.

### 4.1. Language for Test Purposes Description

We designed the language to be as generic as possible w.r.t. the modelling language used to formalize the system. The language is structured as three different layers: *model*, *sequence*, and *test generation directive*.

The *model layer* is for describing the operation calls and the state properties in the terms of the behavioral model M. This layer constitutes the interface between M and the test purposes, and is the only one that is modelling language dependent. The *sequence layer* is based on regular expressions and allows the description of the shape of test scenarios as sequences of operation calls leading to states that satisfy some state properties. The *test generation directive layer* is used to deal with combinatorial issues, by specifying some selection criteria intended for the test generation tool.

We give the syntax of each layer. An example of a test purpose issued from the IAS case study can be seen in Sec. 4.3.

#### *4.1.1. Model Layer*

The syntax of the model layer is given in Fig. 2. The rule SP describes conditions as state predicates over the variables of M. The rule OP allows for describing the operation calls, either by an operation name indicating which operation is called, or by the token $op meaning that any operation is called, or by $op\{OPLIST} meaning that any operation is called except one from the list OPLIST.

#### *4.1.2. Test Generation Directive Layer*

This part of the language is given in Fig. 3. It allows to specify guidelines for the test generation step. We propose one directive aimed at reducing the search for instantiations of the test purposes.

The rule CHOICE introduces two operators denoted as | and ⊗ for covering the branches of a choice. Let $S_1$ and $S_2$ be two test purposes. Then $S_1 \mid S_2$ specifies that the test generator must generate tests for both $S_1$ and $S_2$. $S_1 \otimes S_2$ specifies that the test generator must generate tests for either $S_1$ or $S_2$. This directive is taken into account by the unfolding function that will be shown in Fig. 10 and explained in Sec. 5.2.

#### *4.1.3. Sequence Layer*

This part of the language is given in Fig. 4. The rule SEQ is the root of the grammar for describing a TP as

```
SEQ      ::=    OP   |  "("  SEQ  ")"  |  OP  "⇝("  SP  ")"
              |  SEQ  "."  SEQ
              |  SEQ  REPEAT
              |  SEQ  CHOICE  SEQ

REPEAT   ::=    "*"  |  "+"  |  "?"
              |  "{"  num  "}"  |  "{"  num  ",}"  |  "{,"  num  "}"  |  "{"  num  ","  num  "}"
```

**Fig. 4.** Syntactic Rules for the Sequence Layer

a regular expression.

A step of a sequence is either an operation call as denoted by `OP` (see Fig. 2) or a subsequence of operation calls that leads to a state satisfying a state predicate, as denoted by `OP ⇝(SP)`.

Sequences can be composed by the concatenation of two sequences, the repetition of a sequence or the choice between two sequences. We use the usual regular expression repetition operators (`*` for zero or many times, `+` for one or many times, `?` for zero or one time), augmented with bounded repetition operators (`{n}` means exactly $n$ times, `{n,}` means at least $n$ times, `{,m}` means at most $m$ times, and `{n,m}` means between $n$ and $m$ times). Notice that using the operators `*` and `+` possibly define infinite sets of tests. To be of practical interest, they will have to be instantiated by the test engineer as explicit numbers some time in the process. Using these operators in a test purpose allows the engineer to postpone this decision.

## 4.2. Semantics of the Test Purposes

The semantics of a test purpose expressed in our language is given as a labelled transition system in Def. 6. The semantics of a test purpose TP is bound to a B abstract machine M that is the specification of the system under test. We say that TP is defined on M. We give a unique name to any transition in a set $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$. The binding between the semantics of TP and M is such that the transitions of the semantics of TP are labelled by the names of the operations of M in $\mathcal{A}_{\mathsf{M}}$, and a state predicate of $\mathcal{P}red_{\mathsf{M}}$ on the variables $x$ of M is associated to any state of the semantics of TP.

**Definition 6 (Semantics of a Test Purpose).** The semantics of a test purpose on a model M is a tuple $\langle Q, q_0, T, \lambda, \gamma, Q_f \rangle$ where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $Q_f \subseteq Q$ is the set of terminating states, $T \in \mathcal{T} \rightarrow (Q \times 2^{\mathcal{A}_{\mathsf{M}}} \times Q)$ is a finite set of labelled transitions that are named and denoted by $t_i \longmapsto q_{i-1} \stackrel{op_i}{\rightarrow} q_i$, $\lambda \in Q \rightarrow \mathcal{P}red_{\mathsf{M}}$ is a total function that associates a state predicate, denoted by $\lambda(q_i)$, with every state, and $\gamma \in Q \nrightarrow \{|, \otimes\}$ is a partial function that associates with every source state of a choice expression its kind of operator.

To lighten the vocabulary, in the remainder of the paper, the word test purpose is used both for designing a test purpose expressed in our language, and for designing its semantics.

**Definition 7 (TP Trace).** A finite sequence of transitions $\tau_{\mathsf{TP}} = t_1; t_2; \ldots; t_n$ is a trace of a test purpose TP if there are $q_i \in Q$ and $op_i \in \mathcal{A}_{\mathsf{M}}$ such that for any $i \in 1..n$, $t_i \vdash\!\!\rightarrow q_{i-1} \stackrel{op_i}{\rightarrow} q_i \in T$ and $q_n \in Q_f$.

Given a trace $\tau_{\mathsf{TP}}$, there are zero or many executions of $\tau_{\mathsf{TP}}$ on the B abstract machine on which TP is defined.

**Definition 8 (TP Execution).** Let $\mathsf{M} = \langle x, I, \mathrm{Init}, OP \rangle$ be a B abstract machine. Let $\tau_{\mathsf{TP}} = t_1; t_2; \ldots; t_n$ be a trace of a test purpose $\mathsf{TP} = \langle Q, q_0, T, \lambda, \gamma, Q_f \rangle$ defined on M. $\sigma_{\mathsf{TP}} = (t_1(v_1), w_1); (t_2(v_2), w_2); \ldots; (t_n(v_n), w_n)$ is an execution associated to $\tau_{\mathsf{TP}}$, denoted by $\sigma_{\mathsf{TP}} \in Exec_{\mathsf{TP}}(\mathsf{M}, \tau_{\mathsf{TP}})$, if there are a sequence of state values of TP $q_0; q_1; q_2; \ldots; q_n$, a sequence of state variable values of M $u_0; u_1; u_2; \ldots; u_n$, a sequence of status words values $w_1; w_2; \ldots; w_n$ and a sequence of parameter values $v_1; v_2; \ldots; v_n$ such that:

- $[x' := u_0] prd_x(Init)$,
- for any $i \in 1..n$: $t_i \vdash\!\!\rightarrow q_{i-1} \stackrel{op_i}{\rightarrow} q_i \in T$,
- for any $i \in 1..n$: $[p_i := v_i] T_i(p_i) \wedge [x, x', sw_i, p_i := u_{i-1}, u_i, w_i, v_i] prd_x(S_i) \wedge [x := u_i] \lambda(q_i)$.

We have defined in Def. 6 the semantics of a test purpose as a labelled transition system. We obtain it as follows. We first express the regular expressions as normal forms, based on the three following basic operators: *concatenation*, denoted by ".", *choice* denoted by "|" or "⊗", and *repeat* denoted by "*". The other repetition

operators are redefined from these three basic operators. The instances of the constructions "$\$\mathsf{op}$", "$\$\mathsf{op}\backslash\{$" OPLIST "$\}$" and OP "$\rightsquigarrow($" SP "$)$" are collected as they are, into a set $L$ of atomic symbols. Second, from these normal forms, we compute an automaton $\langle Q, q_0, T', \gamma, Q_f\rangle$ where $T'$ is a set of labelled transitions in the set $Q \times (\mathcal{A}_{\mathsf{M}} \cup L) \times Q$ and $\gamma$ is a partial function in $Q \nrightarrow \{|, \otimes\}$. We apply the usual transformation rules of a regular expression into an automaton to get it. There is however a little difference with the usual rules due to our two choice operators: with $\gamma$, we label the state on which the choice occurs with the corresponding choice operator. Third, assuming that the name $t$ of every transition in $T$ is unique, we transform the automata $\langle Q, q_0, T', \gamma, Q_f\rangle$ that we have obtained into transition systems $\langle Q, q_0, T, \lambda, \gamma, Q_f\rangle$ as follows.

Let *ops* be an OPLIST, $a$ be an operation name in $\mathcal{A}_{\mathsf{M}}$, $b$ be an operation name in $\mathcal{A}_{\mathsf{M}} \cup \{\$\mathsf{op}\}$ and $sp$ be a *state_predicate*:

- $t \longmapsto q \overset{\mathcal{A}_{\mathsf{M}}}{\rightarrow} q' \in T$ if $q \overset{\$\mathsf{op}}{\rightarrow} q' \in T'$ or $q \overset{\$\mathsf{op} \rightsquigarrow (sp)}{\rightarrow} q' \in T'$,
- $t \longmapsto q \overset{\mathcal{A}_{\mathsf{M}} \backslash \{ops\}}{\rightarrow} q' \in T$ if $q \overset{\$\mathsf{op} \backslash \{ops\}}{\rightarrow} q' \in T'$,
- $t \longmapsto q \overset{a}{\rightarrow} q' \in T$ if $q \overset{a}{\rightarrow} q' \in T'$ or $q \overset{a \rightsquigarrow (sp)}{\rightarrow} q' \in T'$,
- for every state $q' \in Q$, $\lambda(q') = \bigwedge_{(q_i \in Q \text{ and } q_i \overset{b \rightsquigarrow (sp_i)}{\rightarrow} q' \in T')} sp_i$ ; otherwise $\lambda(q') = true$.

A test purpose $\mathsf{TP}$ defines a set of finite traces that represents a set of symbolic test cases. We call each trace a $\mathsf{TP}$ trace (see Def. 7). A $\mathsf{TP}$ trace is a finite sequence of transitions that is well formed w.r.t. the transition relation of $\mathsf{TP}$. To be precise, let us notice that it is actually one of the set of sets of finite traces, due to the test generation directive represented by the function $\gamma$ and the operator $\otimes$. For example, the semantics of the regular expression $(a \mid b).(c \otimes d)$ is one of the four following sets of $\mathsf{TP}$ traces: $\{a.c, b.c\}$, $\{a.d, b.d\}$, $\{a.c, b.d\}$ or $\{a.d, b.c\}$. These symbolic test cases must be instantiated as test cases (non symbolic), called $\mathsf{TP}$ executions (see Def. 8) by a symbolic animator from a behavioral model $\mathsf{M}$ and some coverage criteria. In Def. 8, an execution is a finite sequence of pairs made of an operation call provided with the values of its parameters, and the expected status word value returned by the operation call.

The executions are easy to compute by a test generator when the $\mathsf{TP}$ traces are sequences of transition names whose labels have all been instantiated, i.e. in which there is no $\$\mathsf{op}$ label on the transition. Backtracking may be necessary to satisfy the constraints set by the predicates for the states to reach, and the enabling conditions of the operations.

As for the B executions, several $\mathsf{TP}$ executions can be associated to a $\mathsf{TP}$ trace for the same reasons. But in the $\mathsf{TP}$ executions, every operation call $op_i(v_i)$ must moreover lead to a state that satisfies the target state predicate $\lambda(q_i)$ which is associated to the target state $q_i$ of the test purpose. For that, in Def. 8, we have added the following condition for any $i$: $[x := u_i]\lambda(q_i)$. Consequently, it is also possible that no execution is associated to a $\mathsf{TP}$ trace if there is no sequence $u_1; u_2; \ldots; u_n$ of state variable values that satisfy the sequence $\lambda(q_1), \lambda(q_2), \ldots, \lambda(q_n)$ of target state properties.

### 4.3. Test Purpose Example

Here, we exhibit one of the test purposes written for the experimentation of our approach. We wanted to test a property saying that "*to access an object protected by a PIN code, the PIN must be authenticated*". We have written a test purpose that causes the loss of the PIN authentication in all possible ways, and then tries to access the object. The test purpose is given in two stages: the initialization stage and the core testing stage.

Figure 5 presents the initialization stage of the test pattern in four steps, aiming at building the data structure required on the card to run the test. The DF `file_01` and `file_02` and the PIN `pin_02` are names of objects that are defined in the description of the $\mathsf{TP}$. Their types are defined from the types of parameters that they instantiate. Notice that the target state predicates are expressed in the test purpose as B predicates over the objects declared in the $\mathsf{TP}$ and the state variables of the B model $\mathsf{M}$ (see Sec. 3 for the explanation of the variables used in this example). The aim of the first step is to create a new DF denoted `file_01`. The second step aims at creating a PIN object denoted `pin_02` into the DF `file_01` and gaining an authentication over it. The aim of the third step is to create the DF `file_02` into the DF `file_01`. Finally, the last step aims at setting the current DF to `file_01` in order to start the core of the test. The resulting data structure is that of the dashed circled part of the Fig. 1: the DF `file_02` is protected by the PIN `pin_02` for all commands.

```
 CREATE_FILE_DF
    ⇝ (rule_2_obj[{file_01}] ={always} ∧ current_DF = file_01)                    // P1
. PUT_DATA_OBJ_PIN_CREATE . VERIFY
    ⇝ (PIN_2_dfParent(pin_02) = file_01
       ∧ file_01 ∈ pin_authenticated_2_df[{pin_02}])                              // P2
. CREATE_FILE_DF
    ⇝ (rule_2_obj[{file_02}] = {pin_02} ∧ current_DF = file_02)                   // P3
. SELECT_FILE_DF_PARENT
    ⇝ (current_DF = file_01)                                                      // P4
```

**Fig. 5.** Example of a test purpose — initialization stage

```
. (VERIFY | CHANGE_REFERENCE_DATA
| (RESET . SELECT_FILE_DF_CHILD) | RESET_RETRY_COUNTER
| (SELECT_FILE_DF_PARENT . SELECT_FILE_DF_CHILD))
    ⇝(current_DF = file_01 ∧ file_01 ∉ pin_authenticated_2_df[{pin_02}])          // P5
. SELECT_FILE_DF_CHILD
    ⇝(current_DF = file_02)                                                       // P6
. CREATE_FILE_DF|DELETE_FILE | ACTIVATE_FILE | DEACTIVATE_FILE
| TERMINATE_FILE_DF | PUT_DATA_OBJ_PIN_CREATE
```

**Fig. 6.** Example of a test purpose — execution stage

We have given in Fig. 5 and Fig. 6 a label to each target state predicates, so we can refer to it afterwards. These labels appear as double slashed comments on the right hand of each predicate: `// P1`, `// P2`, etc.

Figure 6 shows the core testing stage, describing the test purpose of a successful authentication after all possible ways to lose an authentication. First, the pattern describes the five possible ways for losing the authentication over the PIN `pin_02` (for instance, a failure of the `VERIFY` command or a reset of the retry counter). The aim of the second step is to select the DF `file_02`, with the command `SELECT_FILE_DF_CHILD`. The final step of the test pattern describes the application of six commands, with the current directory file being `file_02` in order to test the correctness of the access conditions.

The complete test purpose is represented as an automaton in Fig. 7. The edges are labelled by the operation names of the pattern and the labels in the vertices refer to the target state predicates Pi of Fig. 5 and Fig. 6. Predicate `true` denotes a state that is not constrained.

## 5. Model-Based Testing Processes

This section first describes a model-based black-box testing process using static structural selection criteria to compute tests from a model. Then we complete this process by using a dynamic selection criterion (TP) instead of static ones, to compute additional tests. This approach is implemented within the Leirios Test

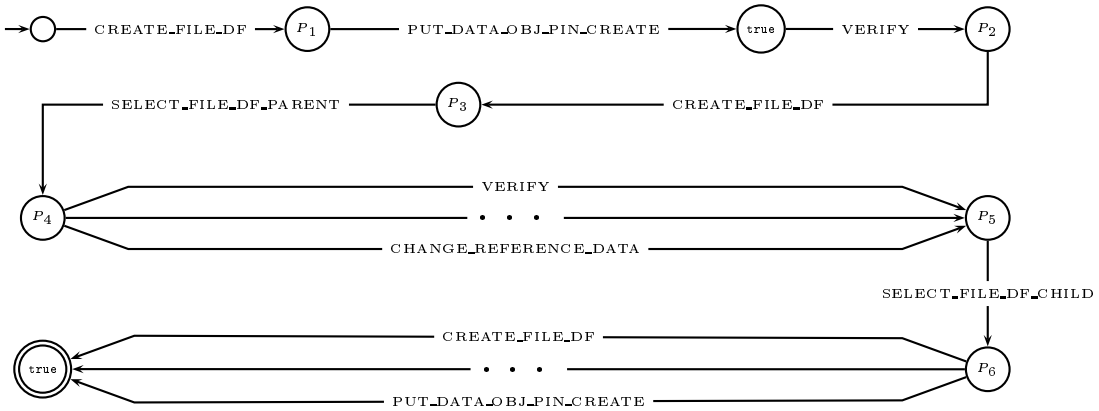


**Fig. 7.** Automaton associated to the test purpose example

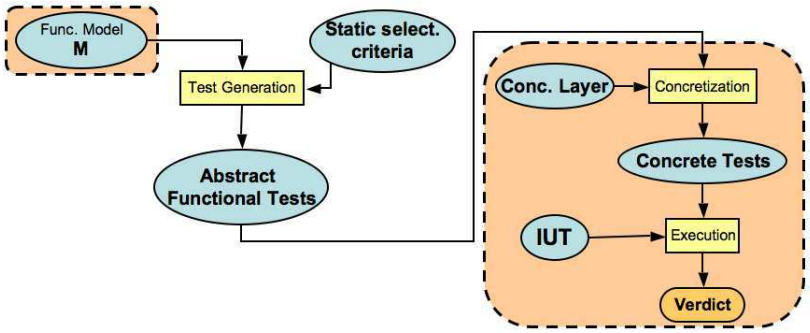


**Fig. 8.** Functional Model-Based Test Generation Process

Generator (LTG) tool [JL07] from Smartesting, that takes B models as inputs. The LTG test computation algorithm, presented in [CLP04], is based on structural coverage criteria of the operations of the model.

## 5.1. Model-Based Testing with Static Selection Criteria

### *5.1.1. Model-Based Testing Process*

The process for computing model-based functional tests is summarized by Fig. 8. The process is made of three steps.

- *Test Generation.* A set of *functional tests* is first statically computed from a behavioral functional model M according to some static selection criteria. In our case, the test generation is performed by LTG. The tool computes test targets from the model according to control flow, decision, condition and data coverage criteria, as further detailed in Sec. 5.1.2 and Sec. 5.1.3.
- *Concretization.* As the tests computed have the abstraction level of the functional model M, they have to be transformed into *concrete tests*, at the level of the implementation under test (IUT). This step relies on the concretization layer which maps the operations and data of M to the operations and data of the IUT, as further explained in Sec. 5.1.4.
- *Execution.* In this step the verdict is given by the comparison between the outputs predicted by M as included in the concrete tests, and the outputs given by the execution of the IUT on the data appearing in the concrete tests (see Sec. 5.1.4).

The dashed circled parts in Fig. 8 show what in the process will be reused to generate tests from dynamic selection criteria (TP), in addition to the functional ones. This will be performed by replacing the abstract functional tests entering the right hand dashed circled part by abstract dynamic tests generated from a functional model M and a TP as it is shown in Fig. 10.

The next three sections detail the composition of the test cases, the generation of test targets by application of static coverage criteria and finally the concretization of test sequences into executable scripts.

### *5.1.2. Test Case Composition*

The purpose of the model-based testing approach of LTG is to activate the operations of the B model. More precisely, it focuses on a path-coverage of the control flow graph of the operations, in which each path is called a *behavior*. Thus, each operation is covered according to its structure, by extracting its nested behaviors. Each behavior is composed of two elements: an activation condition and an effect that describes the evolution of the state variables if the activation condition is satisfied.

For each behavior, a test target is defined as its activation predicate (called decision). The tests covering the behavior will be constituted of a *preamble* that puts the system in a state that satisfies the activation predicate of the behavior. To achieve that, customized algorithms automatically explore the state space defined by the B model and finds one path from the initial state to a state verifying the target. LTG automatically selects the shortest preamble that reaches the test target. It is equipped with a constraint solver and proceeds by symbolic animation to valuate the parameters of a test sequence.

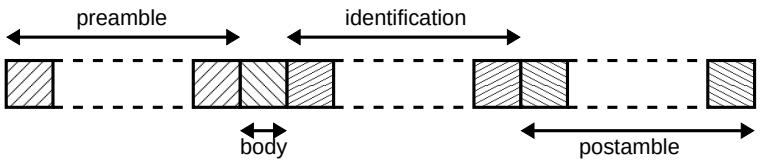


**Fig. 9.** Composition of a LTG test case

Apart from the preamble, a test is thus composed of the 4 elements shown in Fig. 9. The test *body* consists of the invocation of the tested operation with the adequate parameters so that the considered behavior is effectively activated. The *identification* phase is a set of user-defined operation calls that are supposed to perform the observation of the system state. Their invocation when playing the test case on the IUT will make it possible to compare the concretely observed values w.r.t. their expected values computed from the model. Finally, a test case is ended by a *postamble* that is an optional sequence of operations calls that resets the system to its initial state so as to chain the test cases together.

### *5.1.3. Coverage Criteria for Test Target Generation*

From the previous basic definition of a test target, based on the coverage of the structure of the operation model, two other model coverage criteria can be applied, namely predicate and data coverage. These criteria are selected by the validation engineer.

**Predicate coverage** makes it possible to increase the test targets number, and possibly their error detection abilities. This provides a mean for satisfying classical predicate coverage criteria that are: (*i*) *Decision Coverage* (DC) stating that the tests evaluate the decisions (each activation condition) at least once, (*ii*) *Condition/Decision Coverage* (C/DC) stating that each boolean atomic subexpression (called a condition) in a decision has been evaluated as true and false, (*iii*) *Modified Decision/Condition Coverage* (MC/DC) stating that each condition can affect the result of its encompassing decision, or (*iv*) Multiple Condition Coverage (MCC) stating that the tests evaluate each possible combination of satisfying a predicate. In practice, different rewriting rules are applied on the disjunctive predicate form of the decisions, so as to refine the test targets in order to take this coverage criteria into account (for more details see [UL06]).

**Data coverage** makes it possible to indicate which of the test data have to be computed in order to instantiate the tests. The options, applied to operation parameters and/or state variables, propose a choice between: (*i*) all the possible values for a given variable/parameter that satisfy the test target, (*ii*) a smart instantiation that selects a single value for each test data, or (*iii*) boundary value coverage, for numerical data, that will be instantiated to their extrema values (minimal and maximal values).

### *5.1.4. Executable Scripts and Verdicts*

Once the abstract test cases have been computed, they have to be translated into the test bench syntax so as to be automatically executed on the IUT. This is the concretization step.

To achieve that, the validation engineer has to provide two correspondence tables. One of these tables maps the operation signatures of the B model to the control points of the test bench. The other one maps the abstract constant values of the B model to the internal data values of the IUT. By using an appropriate translator, a test script is automatically generated into the syntax of the test bench, ready to be run on the IUT. The correspondence tables and the translator implement the concretization layer.

For each test, the verdict is established by comparing the outputs of the system in response to inputs sent as the successive operations. The concretization layer is in charge of delivering the verdict, by implementing functions that perform the comparison. In this context, the more observation operations (identification phase of Fig. 9) are available, the more accurate the verdict is.

**Limitations** This approach aims at ensuring that the behaviors described in the model also exist in the IUT, and their implementation conforms to the model. Nevertheless, this approach suffers from several limitations.

First, the preamble computed by LTG is always the shortest path from the initial state to the test target. As a consequence, possibly interesting scenarios for reaching this target may be avoided. This implies a lack

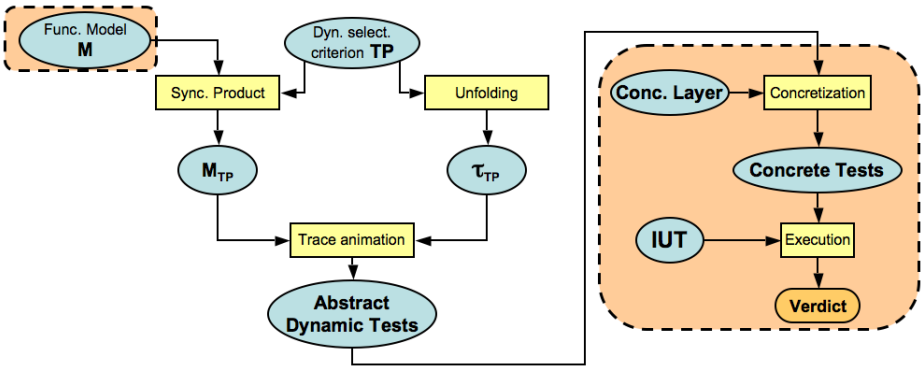


**Fig. 10.** Process for Generating and Executing Tests from a B model and a Test Purpose

of variety in the composition of these preambles, which may avoid revealing errors. Second, the preamble computation is bounded in depth and/or time. This may prevent a test target from being reached.

To overcome these limitations, we now present a model-based testing approach that consists of using dynamic selection criteria to compute new tests w.r.t. the ones issued from LTG.

## 5.2. Model-Based Testing with Dynamic Selection Criteria

Our process for generating tests uses a test purpose TP as selection criterion and a B behavioral functional model M as oracle. The complete process is described by Fig. 10. Notice that the dashed circled parts are the same as in Fig. 8, showing what is reused from the previous process. Here we replace the computation of abstract functional tests based on static selection criteria, by a computation of abstract dynamic tests based on a TP. The abstract dynamic test computation is made in three steps:

- synchronize M and the semantics of TP in $M_{TP}$,
- compute the set of TP traces $\tau_{TP}$ unfolding the semantics of TP,
- compute the set of TP executions (abstract dynamic tests) from $M_{TP}$ and the set of TP traces.

Computing the abstract test cases is obtained by a symbolic animation of the TP traces on a B machine $M_{TP}$ that is the synchronized product between the B model M and the test purpose TP. The synchronized product between M and TP is computed according to the expression in B that is given in Sec. 6. The result is a B machine $M_{TP}$ whose executions are the possible executions from M that conform to TP. Besides, TP is unfolded as a finite set of TP traces (see Def. 7) $\tau_{TP}$, i.e. as sequences of transition names (each one labelled with an un-parameterized operation call) defined according to TP, but without the target states. This set computes all the TP traces whose last state is terminating, and whose length is lower or equal to a maximum length defined by the tester.

We use LTG to instantiate the TP traces. LTG is also a B trace (see Def. 4 in Sec. 2) animator, used by the test engineer to validate its models and manually complete the tests sequences. A TP trace is a B trace of $M_{TP}$. LTG proceeds by symbolic animation. Notice that any other tool with similar capabilities could be used for that purpose. The principle is to "guess" values for the parameters of the operations that make it possible to execute the sequence of operations as described by a particular trace $\tau_{TP}$ of the test purpose TP. In other words, TP executions are computed by LTG animation capabilities from TP traces and $M_{TP}$. The parameter values are computed in LTG by a constraint solver, that finds some values that make the sequences of operations of $\tau_{TP}$ reach the target states given in the TP. No execution is computed when the target states are impossible to reach. The status words are also computed as expected by $M_{TP}$ for these parameters. Additionally, from one TP trace $\tau_{TP}$, LTG will try to compute a TP execution.

The tests computed by this procedure have the abstraction level of the model M of the system and must be concretized as explained in Sec. 5.1.1 in the item entitled *Concretization*.

```
MACHINE M
VARIABLES x
INVARIANT I
INITIALISATION Init
OPERATIONS
    ...
    sw_i ← op_i(p_i) =
        PRE T_i(p_i) THEN S_i END
    ...
END
```

```
MACHINE M_TP
INCLUDES M
SETS Q = {q_0, ..., q_n}
VARIABLES Cq
INVARIANT Cq ∈ Q
    /* Cq : current state of TP */
INITIALISATION Cq := q_0
OPERATIONS
    /* for any t_i ⟼ q_{i-1} -op_i-> q_i ∈ T */
    /* we define an operation t_i s.t. */
    ...
    sw_i ← t_i(p_i) =
    PRE T_i(p_i) THEN
        SELECT Cq = q_{i-1} ∧ ∃(x', sw'_i) ·
                                (prd_x(S_i) ∧ [x := x']λ(q_i))
        THEN sw_i ← op_i(p_i) || Cq := q_i
        END
    END;
    ...
END
```

**Fig. 11.** Combination of a model M and a test purpose TP on M

## 6. Combining a Model and a Test Purpose for Dynamic Selection of Tests

In Fig. 11, we define how to express in B the synchronized product $\mathsf{M}_{\mathsf{TP}}$ of a behavioral model M described as a B abstract machine, and a test purpose TP on M. $\mathsf{M}_{\mathsf{TP}}$ includes the abstract machine M so that it can read the state variables $x$ of M, and it can synchronize any transition $t$ of TP with a call to an operation of M labelled by $t$. The variable $Cq$ represents the current state reached by the last transition executed in the test purpose TP. The initial state is $q_0$. For any transition $t_i$ (such that $T(t_i) = q_{i-1} \stackrel{op_i}{\rightarrow} q_i$), we define an operation also called $t_i$ in $\mathsf{M}_{\mathsf{TP}}$. Its parameter values must satisfy the typing predicate $T_i(p_i)$ of the operation $op_i$ that is called by $t_i$. This operation is enabled if the current state is $q_{i-1}$ and if there are state variable values $x'$ and a status word value $sw'_i$ after $t_i$ that satisfy the before-after predicate of the body of the operation $op_i$ and the target state predicate of the test purpose $\lambda(q_i)$. When these conditions hold, the operation $t_i$ calls the operation of M $op_i$ and places the system in the target state $q_i$ of the test purpose.

Theorem 1 establishes the soundness of the method. For a TP trace $\tau_{\mathsf{TP}} = t_1; t_2; \ldots; t_n$ (see Def. 7), any B execution (see Def. 5) of the B composed abstract machine $\mathsf{M}_{\mathsf{TP}}$ for the B trace $\tau_{\mathsf{M}_{\mathsf{TP}}} = \mathrm{Init}_{\mathsf{M}_{\mathsf{TP}}}; t_1; t_2; \ldots; t_n$ is a TP execution (see Def. 8) of $\tau_{\mathsf{TP}}$ on the abstract machine M. Theorem 2 establishes the method completeness.

**Theorem 1 (Soundness).** Let $\mathsf{M}_{\mathsf{TP}}$ be the B composition of a B model M and a test purpose TP on M as in Fig. 11, and let $\tau_{\mathsf{TP}}$ be a TP trace then,

$$Exec_B(\mathsf{M}_{\mathsf{TP}}, Init_{\mathsf{M}_{\mathsf{TP}}}; \tau_{\mathsf{TP}}) \subseteq Exec_{\mathsf{TP}}(\mathsf{M}, \tau_{\mathsf{TP}}).$$

*Proof.* The proof relies on the fact that, the difference between the B executions of the model M and the TP executions of M, is that, the target predicate $\lambda(q_i)$ holds in every target state $q_i$ of the TP execution. This condition is also satisfied in the B execution of $\mathsf{M}_{\mathsf{TP}}$ since we add this condition in the guard of its operations $t_i$ (see Fig. 11). Moreover, it is obvious that the B executions of $\mathsf{M}_{\mathsf{TP}}$ and the TP executions of M compute the same sequence of states as TP, and execute the same sequence of operation calls as M. □

**Theorem 2 (Completeness).** Given a B composition $\mathsf{M}_{\mathsf{TP}}$ of a B model M, a test purpose TP on M and a TP trace $\tau_{\mathsf{TP}}$,

$$Exec_{\mathsf{TP}}(\mathsf{M}, \tau_{\mathsf{TP}}) \subseteq Exec_B(\mathsf{M}_{\mathsf{TP}}, Init_{\mathsf{M}_{\mathsf{TP}}}; \tau_{\mathsf{TP}}).$$

The proof is straightforward.

Our implementation with LTG computes the B execution of $\mathsf{M}_{\mathsf{TP}}$ with the semantics given in Def. 5. It is sound, but not complete because the constraint solving algorithm is time limited.

| Test purpose | ♯ operations | ♯ transitions | ♯ states |
|---|---|---|---|
| TP1 | 12 | 13 | 12 |
| TP2 | 10 | 17 | 14 |
| TP3 | 9 | 15 | 12 |

**Table 1.** Test purposes description

## 7. Experimental Results

In this part, we report and comment the results of an experimentation done with a security-based B model of IAS, which is 1032 lines long and contains 12 B operations and 19 states variables. This model focuses on access control, and in particular on user authentication by means of a PIN code.

In Sec. 7.1, we present the goal of our experiments. We deduce from this objective the criteria that we must evaluate to reach it. Then we propose an experimental protocol. In Sec. 7.2, we present the experimental results, and we conclude in Sec. 7.3 with the analysis of the results.

### 7.1. Goal, Means and Process of Experimentation

The goal of our experimentations is to answer the question of the complementarity of the test cases generated from dynamic selection criteria, w.r.t. the test cases generated from static selection criteria. We have to address two points to reach this goal:

- we need sets of test cases generated either with dynamic or static selection criteria,
- we need coverage evaluation criteria in order to compare the different test suites.

As for the first point, we have generated four test suites (see Table 2) named LTG, TP1, TP2 and TP3. The LTG test suite have been generated using C/DC static selection criteria with the tool LTG. The three other test suites have been generated using dynamic selection criteria in the shape of three test purposes named TP1, TP2 and TP3. Table 1 gives the number of operations, the number of transitions and the number of states of each test purpose. The first test purpose, that is defined in Sec. 4.3, aims at producing test sequences combining different ways to lose the authentication over a PIN code with the launching of different commands protected by this PIN code. The second test purpose aims at validating the correct interpretation of an access rule, in a context where a confusion could occur between two different PIN objects, due to the complexity of the IAS object reference mechanisms. The third test purpose aims at checking the behavior of the application when an authentication over a PIN object is combined with file life cycle changes.

As for the second point, we have decided to evaluate the coverage of each test suite with respect to a common frame of reference. Directly taking the IAS model as a reference for comparing the coverage of the test campaigns woul not have been a good choice for two reasons: first, the number of states and transitions is too big and second, the part covered by a particular test purpose would be too weak to give significant results. Thus, we have decided to generate an abstraction of the model by focusing on variables giving a good point of view of the states of the system targeted in the test purposes. This abstraction has been computed by the GeneSyst tool [BPS05]. This tool computes a symbolic labelled state-transition system from a B model and the description of the symbolic states that we want to observe, i.e. the domain decomposition of the chosen variables. In our case, the graph produced for IAS was made of 18 states and 497 transitions.

In order to obtain an abstraction which is relevant with respect to the observation of the system, and in particular the access control based on user authentication by means of a PIN code, we have chosen three variables. These variables are: `current_DF` that models the location of the current directory; `df2_dfParent`[5] that represents the structure of the directory tree; and `pin_authenticated_2_df` that indicates the authentication status of a PIN code inside a DF. This choice of variables gave us an abstraction well suited to the observation of the coverage of the tests produced with the test purposes TP1 and TP2. But this abstraction is not well suited to study the coverage of the tests generated from the test purpose TP3. This is due to the fact that TP3 aims at testing the combination of the authentication mechanism with file life cycle changes. The variable representing the file life cycle state has not been taken into account to produce the abstraction,

[5] This function associates each directory with his parent.

| Tests | ♯ tests | Average length | Min length | Max length |
|---|---|---|---|---|
| LTG | 65 | 2.5 | 1 | 5 |
| TP1 | 35 | 9.4 | 9 | 10 |
| TP2 | 66 | 9.5 | 8 | 11 |
| TP3 | 88 | 6.9 | 5 | 8 |

**Table 2.** Test generation results

| Tests | ♯ tests | State coverage | Transition coverage |
|---|---|---|---|
| LTG | 65 | 5/18 = 27.78 % | 33/497 = 6.64 % |
| TP1 | 35 | 9/18 = 50.00 % | 35/497 = 7.04 % |
| TP2 | 66 | 12/18 = 66.67 % | 52/497 = 10.46 % |
| TP3 | 88 | 5/18 = 27.78 % | 23/497 = 4.63 % |
| TP123 | 189 | 13/18 = 72.22 % | 87/497 = 17.51 % |

**Table 3.** Test suites coverage measures

because it resulted in too many symbolic states. It could be interesting to produce another abstraction to study the coverage results of the tests produced with the test purpose TP3.

## 7.2. Results of Test Generation and Comparison of the Test Suites

Tables 2, 3 and 4 give the results of our experimentations. We consider the following test suites:

- the LTG test suite, where tests have been generated using behavior coverage criteria with coverage of conditions and decisions (C/DC) and coverage of boundary values for the operation parameters;
- the three test suites TP1, TP2 and TP3, where tests have been generated using respectively the test purposes TP1, TP2 and TP3 as dynamic coverage criteria.

Table 2 indicates for each test suite the number of tests computed, the average number of operation calls per test sequence and the minimal and maximal number of operation calls per test sequence.

Table 3 presents the state and transition coverage achieved by each test suite as well as by the union of the three test suites generated using the test purposes.

The complementarity of a test suite $e_1$ w.r.t. a test suite $e_2$ is denoted as $comp(e_1, e_2)$. We measure it as the ratio between the number of transitions covered solely by $e_1$ (i.e. not by $e_2$) and the full number of transitions covered by $e_1$ (possibly including transitions also covered by $e_2$). If $cov(e)$ is the number of transitions covered by a test suite $e$, then $comp(e_1, e_2) = \frac{cov(e_1 \cup e_2) - cov(e_2)}{cov(e_1)}$.

We need additional coverage results given in Table 4 to measure the complementarity of the test suites issued either from LTG or from the test purposes:

- TP1 ∪ LTG, TP2 ∪ LTG and TP3 ∪ LTG give the coverage achieved by the union of each test suite issued from the test purposes with the LTG test suite;
- TP123 ∪ LTG gives the coverage achieved by the union of all the test suites.

The last two columns of Table 4 give the percentage of transitions that are not redundantly covered by the test suites of LTG and by the ones issued from the test purposes.

| Test suite | ♯ tests | State coverage | Transition coverage | $comp(LTG, TP_i)$ | $comp(TP_i, LTG)$ |
|---|---|---|---|---|---|
| TP1 ∪ LTG | 100 | 9/18 = 50.00 % | 63/497 = 12.68 % | 28/33 = 84.8 % | 30/35 = 85.7 % |
| TP2 ∪ LTG | 131 | 12/18 = 66.67 % | 83/497 = 16.70 % | 31/33 = 93.9 % | 50/52 = 96.2 % |
| TP3 ∪ LTG | 153 | 6/18 = 33.33 % | 51/497 = 10.26 % | 28/33 = 84.8 % | 18/23 = 78.3 % |
| TP123 ∪ LTG | 254 | 13/18 = 72.22 % | 109/497 = 21.93 % | 22/33 = 66.7 % | 76/87 = 87.4 % |

**Table 4.** Measures of the Complementarity of the Transitions Covered

### 7.3. Report and Conclusion About the Results

The coverage evaluation corroborates the fact that the tests generated using test purposes as dynamic coverage criteria complement the tests generated using static criteria.

Table 2 shows that the average length of the tests generated from the test purposes is between 2.7 and 3.8 times longer than the tests generated from static selection criteria. Table 3 shows that the tests generated from the test purposes cover up to twice as many states and transitions than the tests generated from static selection criteria.

The first part of Table 3 shows that the test suites obtained from test purposes give a better coverage of the states and transitions of the abstraction than LTG, except for the last test purpose TP3. The better coverage –such as 66.67 % of states and 10.46 % of transitions for the tests generated using TP2– is due to the fact that test purposes were designed to test the access control, and that the abstraction has been chosen to focus on it. The poor coverage results obtained with the third test purpose are due to the fact that the abstraction was not suited to TP3 (see Sec. 7.1).

The results given in Table 4 clearly show that there is little redundancy between the tests issued from LTG and the ones issued from the test purposes. Nearly 85% and more of the transitions covered by the LTG tests are not covered by the test purposes ones, and *vice-versa*. There are two slightly lower ratios. "Only" 66.7% of the LTG tests differ from the union of the ones issued from TP1, TP2 and TP3. This is not surprising since the intersections of the LTG tests with each of the three test purposes are put together by this measure. We also see that less than 80% of the TP3 tests are complementary to the LTG ones. This comes again from the abstraction not well suited to TP3. Nevertheless, the ratio (78.3%) remains good. Finally, we think that *All-Transition-Pairs* coverage criterion (every pair of adjacent transitions in the state transition model must be traversed at least once), which has not been studied in this paper, could also serve our intention to show the complementarity of the different test suites.

These results show that we have increased the coverage of the system –in particular, the access control part which is observed by the abstraction– by generating test suites from the three different test purposes. These results also show that the test purposes that we designed lead to complementary test sequences w.r.t. the tests generated from static selection criteria.

## 8. Conclusion

We have presented in the B framework a method for generating tests from test purposes in a behavioral model-based testing context. We have performed experiments on the industrial smart card platform IAS. This experimentation shows that the tests that we have generated are complementary w.r.t. the structural ones [BLLP04, SLB05]. The method makes use of already existing material, written for model-based structural testing: the behavioral model, the concretization layer and the test execution environment. The approach also re-uses the set theory constraint solvers and the algorithms for preamble searching of a test target. Additionally, test purposes are written by a test engineer to describe his test intentions.We have presented a language dedicated to the expression of the test purposes. The language allows the tester to describe operations to be called as well as states to be reached. Writing a test purpose needs good expertise in the model of the system on behalf of the tester. He must express the set of executions for which he wishes a test selection by a test purpose. But the expressivity of the language that we propose makes their descriptions easier, thanks for example to the use of regular expressions. In general, it would be far more difficult, if possible at all, to drive the static generator by transforming the behavioral model and/or adapting the static selection criteria, in such a way that it finds similar tests to the ones generated from test purposes.

The method easily ensures the traceability of the tests generated to the original test purposes, since the tests are computed from them. Also, with the traceability mechanism for functional test generation that we use, we know which operation behaviors have been covered.

Among the works on Model-Based Testing, some use static (or structural) test selection criteria [EFHP02, BLLP04, UL06], applied to the behavioral model. Some other works apply dynamic criteria. Our works fit in this second category, and complete a test generation environment based on static criteria. Dynamic selection criteria target specific classes of execution of the system. The aim is to test dynamic properties such as safety properties, security properties (access control [DJM08, PMLT08], integrity, authentication, etc.), and partial availability properties called possibilities in [CJMR07]. In the previous cited works, dynamic selection

criteria are described as input-output labelled transition systems. We have called test purposes these dynamic selection criteria.

Many other works use test purposes as selection criteria to extract tests from a model. The test purposes are described by temporal properties in a temporal logic [ADX01, TSL04], input output Labelled (or Symbolic) Transition Systems ioLTS (ioSTS [JJRZ05, CIVDP07, FTW05]), or use cases [GHN93].

As in all these approaches, our method performs the synchronized product between the test purpose and a behavioral model. Two points make our method different from the approaches with properties expressed as temporal logic formulas. On one hand, the test purposes express a test intention from the tester by a combination of state sequencing (as in temporal logic) and operation calls (which does not exist in temporal logic). On the other hand, the test generation technology is different. Temporal logic based approaches use model-checkers, that generate tests by exhibiting counter-examples. Our approach uses constraint solving techniques to perform symbolic executions, on symbolic values of the parameters of the operations. Thus it is possible to treat infinite data domains, thanks to strategies of static selection of finite sets of representatives. Finally, the approach [ADX01] uses property mutation techniques, based on syntactical transformation of operators. In our approach, the tester combines a test need with a property, which can be seen as a semantic mutation of a property. Our mutations introduce modifications in the sequencing of operation calls while the automatic mutations transform the propositional or relational operators used in the atomic conditions.

Our approach differs from approaches such as the one adopted by TGV [JJ05] (resp. STG [JJRZ05]) that use IOLTS (resp. IOSTS) expressing operation calls, with no information on the targeted states. These approaches use constraint solving techniques on data in integer and boolean scalar domains. We also use constraint solvers on more complex data structures of set theory domains, in order to fully treat the behavioral B modelling language (sets, functions, relations and sequences). The approaches with IOSTS also use symbolic execution techniques by abstract interpretation, to reduce the size of the synchronized product. The unreachable states are suppressed by over-approximation. This abstract interpretation allows treating symbolic models. Our approach uses a symbolic model to evaluate the tests coverage.

In [SML06], the authors present a test case generation algorithm from B event systems and use cases by refinement. There are three main differences with our approach. Our method reuses abstract B machines and a concretization layer CL dedicated to the functional test generation. Therefore we do not refine the test cases. Moreover, our test purposes are more expressive use cases that contain target state descriptions.

As a difference with the preceding approaches, we have shown in a previous work [MJP+07] how the test purposes can be automatically computed, by modelling some *test needs* as syntactic transformation rules that transform behavioral properties. We are currently working at identifying and writing such transformation rules, based on the IAS case study. This work needs to be developed by studying many other case studies (for instance, the mini-challenge that proposes to design and verify a POSIX compliant flash-based system [JH07]) in order to produce rules sufficiently generic to be applicable to a variety of examples. Rules could also be automatically deduced from the syntactic expression of a property, as suggested by [BDGJ06] for properties expressed in JTPL, a temporal logic for JML.

The method that we have presented works well, and is applicable to industrial size applications as long as the TPs are not too generic. By that, we mean that the constructions $\$op^+$ or $\$op^*$, although allowed by the language, are not used by the tester. If no $\$op$ is used at all, then all the operation calls are explicitly defined, and we find their parameter values by animation of the behavioral model M. If $\$op$ is used with no repetition operator, it is still easy to instantiate it as an operation call: this is obtained by trying every operation at most once. But when the constructions $\$op^+$ or $\$op^*$ are used, the valuation becomes more complicated. Indeed, every such construction has to be instantiated, i.e. replaced by a sub-sequence of valuated and explicitly defined operation calls. This implies searching amongst all the possible instantiations, one for which there are parameter values that cause the sub-sequence to reach the targeted symbolic state specified in the TP. There is a combinatorial explosion of the possibilities. To deal with this situation, we plan to generate an abstraction of the system, based on variables and sub-domains identified in the TP. We could synchronize this abstraction with the TP. We would thus obtain a view of the system where the generic operation calls have been instantiated. We could use this view to generate tests from a static selection criterion, such as the coverage of the states, or of the transitions of this view. These tests would be symbolic tests, in the shape of a sequence of operation calls, provided with symbolic values of their parameters. They would have to be valuated afterwards from the detailed behavioral model. We could also use the abstraction synchronized with the TP as a reference model to evaluate the tests coverage. This approach raises two technological challenges. On one hand, it is necessary to have a time efficient technology of abstraction, that can be applied in practice. On the other hand, the abstraction techniques can fold back sequences of operation calls into cycles. So, the

search of a valuation of the symbolic tests will have to find sub-sequences of operations to insert between two symbolic calls. But this cycle combination search is highly combinatorial. Thus, the issue will be to find incomplete, but practically efficient, search techniques. This means techniques that provide reasonably good coverage rates for the examples treated.

## References


[Abr96] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ADX01] P. Amman, W. Ding, and D. Xu. Using a model checker to test safety properties. In *ICECCS'01*. IEEE Computer Society, 2001.

[BDGJ06] F. Bouquet, F. Dadeau, J. Groslambert, and J. Julliand. Safety property driven test generation from JML specifications. In *FATES/RV'06*, volume 4262 of *LNCS*, pages 225–239. Springer, 2006.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005.

[BLLP04] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Software: Practice and Experience*, 34(10):915–948, 2004.

[BPS05] D. Bert, M.-L. Potet, and N. Stouls. Genesyst: a tool to reason about behavioral aspects of B event specifications. In *ZB'05*, volume 3455 of *LNCS*, 2005.

[CIVDP07] J. Calamé, N. Ioustinova, and J. Van De Pol. Automatic model-based generation of parameterized test cases using data abstraction. *ENTCS*, 191:25–48, 2007.

[CJMR07] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558–574, August 2007.

[CLP04] S. Colin, B. Legeard, and F. Peureux. Preamble computation in automated test case generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3):213–235, 2004.

[DJM08] J. Dubreil, T. Jéron, and H. Marchand. Automatic test generation for security properties. Deliverable L3.3, INRIA/IRISA Vertecs Project, 2008. Politess Project, ANR-05-RNRT-01301.

[EFHP02] E. E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[FTW05] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004, Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 1–15. Springer, 2005.

[GHN93] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs. In *SDL'93 - Using Objects*, October 1993.

[GIX04] GIXEL. *Common IAS Platform for eAdministration*, Technical Specifications, 1.01 Premium edition, 2004. http://www.gixel.fr.

[JH07] R. Joshi and G. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, June 2007.

[JJ05] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *Software Tools for Technology Transfert*, 7(1), 2005.

[JJRZ05] T. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *TACAS'05*, volume 3440 of *LNCS*, pages 349–364. Springer, 2005.

[JL07] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated test generation from B models. In *B'2007*, volume 4355 of *LNCS*, pages 277–280. Springer, 2007.

[JMT08a] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *AST'08, 3rd Int. workshop on Automation of Software Test*, pages 41–44, Leipzig, Germany, May 2008. ACM Press.

[JMT08b] J. Julliand, P.-A. Masson, and R. Tissot. Generating tests from B specifications and test purposes. In *ABZ'08, Int. Conf. on ASM, B and Z*, volume 5328 of *LNCS*, pages 139–152, London, UK, September 2008. Springer.

[MJP+07] P.-A. Masson, J. Julliand, J.-C. Plessis, E. Jaffuel, and G. Debois. Automatic generation of model-based tests for a class of security properties. In *A-MOST'07*, pages 12–22. ACM Press, 2007.

[PMLT08] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-based tests for access control policies. In *ICST'08, Int. Conf. on Software Testing, Verification, and Validation*, pages 338–347. IEEE Computer Society, 2008.

[SLB05] M. Satpathy, M. Leuschel, and M. Butler. ProTest: An automatic test environment for B specifications. In *MBT'04*, volume 111 of *ENTCS*, pages 113–136, 2005.

[SML06] M. Satpathy, Q.-A. Malik, and J. Lilius. Synthesis of scenario based test cases from B models. In *FATES/RV'06*, volume 4262 of *LNCS*, pages 133–149. Springer, 2006.

[TSL04] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IRI'2004, IEEE Int. Conf. on Information Reuse and Integration*, pages 413–498, November 2004.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006.